W.B. Ean

C++ in ONE DAY

The Ultimate Beginners Guide To C++ With 7 Awesome Projects

C++ In One Day

The Ultimate Beginners Guide To C++ With 7 Awesome Projects

By W.B Ean

Introduction:

Many times we wonder how computers can work from the inside. For the end user, there is practically no interaction with something deeper. They do not need to know many details about how the computer works or something about programming. The end user is only attracted by the functionality and the program to have a simple interface that is easy to handle.

However, the software does not create by itself. Somebody has to do the dirty work. Or, like other like to call it, the art of programming. The main part of computers is their microprocessor, the brain that executes every instruction needed for every task the user needs. These processor only understands binary code (zeros and ones) and it does not know nothing about English or numbers.

So, how can we interact with the processor? We use programming languages. Through history, many languages have been developed for the interaction between the programmers and the machines. In recent times, programming has become a major field of study with a wide variety of programming language to choose depending on the purpose.

One of the most preferred languages is C++. Why? Mainly because of its speed, its standards and the wide range of applications. C++ was developed as an improvement for the C programming language. This change added many features, like enabling object oriented programming, templates, memory handling, error handling and more tools for the programmers.

The C++ in one day Course is targeted for people who want to learn the basics of the C++ programming language. We will cover from the most basic concepts and stuff, perfect for beginners, and as the course goes, we will be adding more features from the language and also start to develop some real life projects.

This course is divided in four sections:

1. The basics of C++: In this chapter, we will cover the basics of C++. This is the main core of the course for beginners. Here we will cover the basic structure for every C++ program. A brief explanation of the library system. The standard system of input and output from C++. Datatypes are another fundamental topic to cover and will be discussed, including examples of

declaration, using and storing values. String processing will be covered.

In this chapter, we will also explain the basic flow structures that are used in C++ for designing software. The if structure, the switch structure, the while structure and the for structure. Finally, a very brief introduction to the basics of object oriented programming.

2. Things to do with C++: In this chapter, we will discuss what kinds of software can be developed using the C++ programming language.
3. Implementing projects with C++: Here we will write some code! 7 projects have been selected and programmed in order to show you how you can do useful things with C++.
4. What to do next: Finally, we finish the course with some recommendations on topics that, while are not very necessary for the basics, they are a good place to follow after learning the basics. These features can enable you to design more optimal and complex software.

So, let’s get started!

Chapter 1: Basics of C++

C++ is a multipurpose programming language. What is this? That it can be used for writing every type of software that the programmer wants. It can be from a simple program like computing the sum of two numbers up to a full operating system (for example, Windows) and videogames (PS3, PS4, XBOX). Some advantages that C++ has over its competitors (like C# or Java) are, in first place, its speed. When you write a C++ program, it is compiled to a binary file that runs along with the operating system. Second, it is standard. C++ has a very well defined standard from beginning to end and this means that, if the programmer writes code in standard C++, it can be ported to other compilers with minimum or small changes.

However, not everything is as simple as its sounds. Although C++ is very fast, it is very difficult to master due to the libraries that it has and the Standard Template Library (STL) that will be discussed in later chapters.

But let's get started with the basics of the language.

Instructions and compiler
Through the article we will be using the GNU C++ compiler, G++, along with the IDE CodeLite. All of the code will be written in standard C++. CodeLite and the compiler can be downloaded from http://downloads.codelite.org/

Libraries
The full extension of C++ has libraries for many purposes: input and output, math, memory management, time, file processing, string processing and many more features. However, when we start to code, we must define which of these libraries we want to use in our program.
Since 2003, C++ also implemented a feature named **namespaces**, which are another way to organize all of the stuff inside the language.

The *main* function In C++, all programs must contain a function (or a piece of code) named **main**. The purpose of this function is to indicate the compiler where will the program start its execution.

Input and Output (I/O)
Almost every program in C++ needs an input and an output. There are some exceptions to the rule, but normally, there is always I/O. The input is all the data that enters the computer, like numbers, words. The output are the data that is showed to the screen (also named console). The library used in C++ for this operations is named **iostream**.
Inside iostream, there are two objects that we will be using to read data from the keyboard and showing data to the console: **cout** and **cin**.

Before programming anything, in CodeLite we must create a new project. Steps for creating a new project in CodeLite:

- In the main screen you will see a button with the text “NEW”. Click it.
- When the new window appears, select on the left side the option “Console” and select “Simple executable (g++)” from the list.
- On the right panel choose a name for your project, for example “Example1”.
- In project path, select a folder in your computer where you will save your code.
- In the compiler section, be sure to have “gnu g++” selected.
- Click OK

After doing this, go to the left side of your screen and will find a folder with the name you selected for your project. Open it and you will see another folder named **src** (abbreviation for **source**). Inside src, there will be a file named **main.cpp**. This is the file where you will write your C++ code.

The example code for a simple input and output in C++ is as following: #include <iostream>

```
using namespace std;

int main()

                                                  {

      cout << "This is my first C++ program"; return 0;

                                                  }
```

As you can appreciate, the first line we included the library for I/O. On the second line, **using namespace std** we say to C++ that we want to use all of the features inside the selected libraries (like **cout**).
After this, we find our main function. Here is where all the execution begins. All functions in C++ hace an opening and closing brace that indicates where the function starts and ends. Inside main, there is the line **cout << “This is my first C++ program”;** This line will output everything inside the double quotes to the console. Notice that this line ends with a semicolon. In C++, all the instructions inside a block of code end with a semicolon (there are some exceptions that will be covered).
You can see that between the **cout** object and our text, there are two < signs. They are named **overloaded operators** and they represent the direction where the information is traveling, in this case the information is going outside the computer. As you may think, when we want to make an input to the program, we will be using the >> operators.
For executing the program, go to the **Build** menu and select the option **Build and Run project.** You will see how a black screen appears. This is the console and this is where we will perform our input and output.

Pausing the execution
If the console appears and immediately closes, don’t worry, it is normal. But we need to see the output. For this, add: cin.get()
Before return 0;

Data Types and variables
When we work with a program, there are many types of data we can use. For example: numbers, text, and

some special types of data. All of these are called Data Types. They are very important in C++ because it is the main point for developing inputs. In C++ we have 3 categories: numeric, alphabetic and binary. In the numeric category we have 3 types of data:

- **int –** Represent integer numbers
- **float** – Represents numbers with 7 decimal positions after the point
- **double** – Represents numbers with 15/16 decimal positions after the point

There are more specific numeric data types but for the purposes of this article, these three are enough. For the alphabetic category we have two types of data:

- **char** – Store one alphanumeric digit
- **string** – Store a sequence of alphanumeric digits

And in the binary category, there is only one data type:

- **bool** – Store a true or false. Zero or One.

The bool data type may sound very useless at first but it has its handy applications when programming some specific routines.

But, why do we need all of these types? Simple: In C++ there is a concept named **variable**. A variable can store information of any of these mentioned types. Imagine a variable like a box that can hold some value for a certain time and you can do any operation with that value.
In C++, before using variables, we must declare them. How? Like this: **<datatype> <nameOfVariable>**

Let’s make a simple example. We want to calculate the sum of two numbers. These numbers are integer. So, in order to accomplish this, we must declare three variables, one for each number and another one for the result:

```
int numberA;
int numberB;
int result;
```

As you can see, for each variable we defined the int type and after the data type, the name for the variable. You can name the variables in the way that you want, but it is recommendable to give them a name similar to the use it will have. Now the question is: How can we assing a value? We use the equal **=** operator:

```
numberA = 10;
numberB = 5;
result = numberA + numberB;
```

For this, I assigned 10 to numberA and 5 to numberB. After that, we assigned the value of result to the sum of both variables. The complete program would look like this:

```
#include <iostream>
using namespace std;

int main()

{

        int numberA;
        int numberB;
        int result;
        numberA = 10;
        numberB = 5;
        result = numberA + numberB;
        cout << result;
        cin.get();
        return 0;

}
```

Notice that here, we used **cout << result;** for our output without the double quotes. When you use double quoutes, ALL the text inside the quotes will go directly to the screen. However, when we want to display the value of a variable, we must write the name of the variable without quotes. C++ then will look for the value inside the variable and show it on your screen.

There are some limitations to this code. For example: we have to assign values directly inside the code for the sum. The optimal way would be giving the user the option for input the values into the program. This can be done like this:

```
#include <iostream>
using namespace std;

int main()

{

        int numberA;
        int numberB;
        int result;

        cout << "Enter first value: ";
        cin >> numberA;
        cout << "Enter second value: ";
        cin >> numberB;

        result = numberA + numberB;
        cout << "The sum of both values is: " << result; cin.get();
        return 0;

}
```

Notice some differences in relation to the other code. First, the declaration of the three variables we need. After this, we tell the user to enter the first value. We can see how **cin>>numerA** is used. With this line, we tell C++ to capture the input from the keyboard and the value assign it to the variable numberA. The same for numberB. After this, the result of the sum is calculated and is sent to the output. Notice how in the last output, the overloaded operation appears twice. This is very convenient we want to show multiple values in the same line.

Now, let's say we want to know the average of these two values. We can modify the result line in the following way: result = (numberA + numberB) / 2;
The parenthesis indicates that the mathematical operations inside the parenthesis will be done firstly. After that, it will continue with the rest of the expression. However, we have a problem with this line. Suppose numberA has the value of 5 and numberB has a value of 6. The result would be 5.5. No problem here until… we notice that result is declared as **int**. And, according to our data types, the int data type is reserved only for integer. So, what should we use? We must change the datatype of result to either **float** or **double**.
double result;

Besides sum and divition, there are more math operators to perform operations. The sustraction is performed with the "-" operator, the product is performed with the "*" operator. And a special case is the "%" operator, named Modulus. The modulus will return the left over of a division.
More complex mathemathical operations can be performed with C++ using the **<complex>** library. Por example, power, square root, trigonometrical functions, and more.

String I/O
The management of string in C++ can be done with the library **<string>**. It includes the object string that we need to use. For example, let's suppose we want a program to say goodbye to anyone. It would look like this:

```
#include <iostream>
#include <string>
using namespace std;

int main()

                                        {

        string name;

        cout << "What's your name? ";
        cin >> name;
        cout << "Goodbye " << name;
        cin.get();
        return 0;

                                        }
```

The first thing you may notice is that we added the string library. Inside main, just like any variable, we declared **name** as **string** type. After that, we requested the user to write their name and then output the sentence "Goodbye" followed by the given name. Let's see another example:

```
#include <iostream>
#include <string>
using namespace std;

int main()

                                        {
```

```
                              {

    string name;
    string lastname;
    string fullname;

    cout << "What's your name? ";
    cin >> name;
    cout << "And your last name? ";
    cin >> lastname;
    fullname = name + " " + lastname;
    cout << "Goodbye " << fullname; cin.get();
    return 0;

                              }
```

In this code, I declared a couple of extra strings. We also request the user their last name. In programming, we call **concatenate** to the action of attach strings. In this example, we concatenate name, an empty space and last name to produce a unique string containing the full name. And that string is sent to the output.

Scape Sequences

This is an addition to the I/O topic. In C++, there are special sequences that allow the programmer a better distribution of the information that is displayed. The sequences are:

- \n – End of line
- \t – Insertion of tabular space

With these two sequences, you can design a well distributed interface. Look at the example: cout << “Here is a line\n” << “Here is another line\t” << “I am after a tab space”; This is a simple example of the sequences and can be used only with the cout object.

Program Flow

Up until this point. We have covered the types of data, how to show info to the screen and capturing data from the keyboard. Now, a very important part of C++ are the flow control instructions. With this part, you will be able to take decisions and make complex programs.

If Structure

The **if** is a conditional structure. Like its own name says, *if something happens, then do this thing*. The structure goes as follow: if(condition)

```
                              {

     // do something

                              }
```

The condition is a expression in which the result will be true or false. There are some operators to evaluate the condition:

- EQUALS (==) : Will be true if both values are the same. Example: a=b
- AND (&&) : Will be true if all values are true.
- OR (||) : Will be true if any of the values are true.
- NOT (!) : Will be true if the value is false.
- GREATER (A>B) : Will be true if A is greater than B
- LESS (A<B) : Will be true if B is greater than A.

Let’s suppose we must evaluate if a student approved or not a course. Look a the following code: #include <iostream>

```
using namespace std;

int main()

                                   {

    int note;

    cout << "Enter your note: ";
    cin >> note;
    if(note>6)

                                   {

        cout << "Congratulations. You passed!"; }
    return 0;

                                   }
```

We ask the user for their course note. If the user enters a number greater than 6, then the flow of the program enters the code block surrounded by the braces. If not, it ignores this block and continues its execution. Notice that the IF instruction does not end with semicolon. This is because it is followed by a block of code. But, what happens when the note is less than six?
We use the **else** keyword.

```
#include <iostream>
using namespace std;

int main()

                                   {
```

```
        int note;

        cout << "Enter your note: ";
        cin >> note;
        if(note>6)

                                {

            cout << "Congratulations. You passed!"; }
        else

                                {

            cout << "Sorry. You did not passed"; }
        return 0;

                                }
```

This code is exactly the same as the last one with one difference. We added the **else** keyword with its respective block of code. First, the condition note>6 is evaluated. If this condition is true, then the block of code next to the if will be executed and the else block code will be ignored. If the condition is false, then the if block code will be ignored and the else block code will be executed.

We can also add multiple conditions in the if clause. Look at the following code: if(note==10)

```
                                {

        cout << "Excellent!";

                            }

    else if(note>6 && note<10)

                                {

        cout << "Congratulations. You passed"; }
    else

                                {

        cout << "Sorry. You did not approve"; }
```

We added something called **else if** and it allows the programmer to add multiple conditions on the same if. In this example, if note equals to ten, only the first block of code is going to be executed. On the second contidion, if note is greater than six AND note is less than 10, then the middle block of code will be executed. You can add as many else if as you may want or need. However, if neither the IF nor any of the ELSE IF conditions are true, then the ELSE block of code will be executed. The else is only executed when none of the IF or ELSE IF are true.

Making easier multiple conditions

You might think “Hey, what if I have a lot of conditions, like 20 or 30? It will be a lot of code” and yes, you are right. It would definitely be a lot of hard work having many conditions. But C++ has a answer for this: the **switch** instruction. With switch, we can use multiple conditions in a simple way. Continuing the same example of the notes, let’s suppose we want a different message for the approving notes. Starting from six, there will be 5 different messages for approving and a single message for failing. Using the switch statement, the code would look like this: switch(note)

```
                {

    case 10:
            cout << "Excellent"; break;
    case 9:
            cout << "Very good note"; break;
    case 8:
            cout << "Good job!"; break;
    case 7:
            cout << "Not bad. Keep working"; break;
    case 6:
            cout << "You passed by a bit"; break;
    default:
            cout << "You did not approve"; }
```

The first thing to watch is that the variable we want to compare goes inside curly braces next to the switch statement. It does not need to end with semicolon because it is followed by the block of code for the switch. For each comparison, we use the **case** keyword followed by the value we are looking. In case note has a value of 10, the message “Excellent” goes to the output. For each case, we must add the **break** keyword. If we do not do this, the switch statement will fall to the next case until either finds a break or the end of the switch code block.

Loops

Alright, so you now know how to take decisions on your program. You can direct the flow to another pieces of code and decide which one execute and when. Excellent. Let’s say that you want to calculate an average. Each student is taking 5 subjects and you want to know the average of those note. Simple, you sum them up and divide by 5. But, there is not one student on the school. Can you copy the code and repeat this 10 or 15 times? Well.. yes, you could do it, but the code would start being very unmanageable. So, how can we fix this problem? C++ has the right tool for the job, and it is called **loops**.

A loop is a block of code that will repeat itself until a condition is satisfied. We have two kinds of loops: counting loops and conditional loops. The first one is called **for** and it is used when we already know how many times we want to repeat the block of code. The syntax for the **for** loop is like this: for(VARIABLES; CONTIDION; INCREMENT)

{

// Repeat some code

}

The second is named **while**, and it will loop until some condition is satisfied. The syntax looks like this: while(CONDITION)

{

// Repeat some code

}

Following the example of the averages, calculating the average for 10 students will look like this: float note1, note2, note3, note4, note5;
float avg;

for(int i = 0; i != 10; i++)

{

cout << "First note: "; cin >> note1; cout << "Second note: "; cin >> note2; cout << "Third note: "; cin >> note3; cout << "Fourth note: "; cin >> note4; cout << "Fifth note: "; cin >> note5; avg = (note1+note2+note3+note4+note5)/5; cout << "Average: " << avg << "\n"; }

As you can see, we declared the variable that we are going to use outside the loop. After this, we declared a variable named **i**. **I** is the counter and it will be incremented every loop. Look at the declaration: it starts at zero. On the condition section, the operator "!=" means **different**, so, the loop continue while until I equals to 10 (that means, 10 times). After this, go to the block of code. The program asks the user to enter five notes. Don't worry about cout and cin on the same line, that is the use of the semicolon, separating sentences. After the five notes are inside, the average is calculated and shown to the display. When we reach the end of the block the code, we go back to the increment and I now equals to I + 1. The ++ operator means that the variable is incremented by one. With this piece of code, we can run the loop for 10 times. But, let's say we want to tell the loop how many students are on the school, not just an antiquated 10-times-run. It could be this way: int limit;

cout << "How many students are in your school? :";
cin >> limit;

for(int i = 0; i != limit; i++)

{ . . . }

Here, we declared another variable named limit. That number is obtained from the keyboard and the loop will continue until I equals to limit.
As you might observe, we must always know when the loop is going to end (either 10 or a variable). But, how we do when we do not know how to stop? This is a situation when the **while** loop is very useful. Let’s say you want to calculate the average BUT you do not know how many subjects the school has. Just keep the program running until you told it to. This could be a very real situation. And we cannot ask the student.
So, look at this piece: int val = 1;

while(val==1)

{

cout << "First note: "; cin >> note1; cout << "Second note: "; cin >> note2; cout << "Third note: "; cin >> note3; cout << "Fourth note: "; cin >> note4; cout << "Fifth note: "; cin >> note5; avg = (note1+note2+note3+note4+note5)/5; cout << "Average: " << avg << "\n"; cout << "Do you want to continue? [1] YES. [Other] NO"; cin >> val;

}

This would be read as follow: “While value is equal to one, execute this code”. When does val changes its value? At the end of the block of code, where we ask the user if he wants to continue or not. If he selects one, val continues having a one. Anything else will result on the termination of the loop.

Infinite loops and exits
Sometimes it’s very useful to have infinite loops. This will not be much covered but it is good to know how.
In the while loop, its: While(true)

{

// Do something

}

For the loop for:
for(;;)

{

// Do something

```
}
```

The keyword used for ending the loops is **break** (yes, the one from switch). It can be used like this:

```
while(true)
{
    if(somethingHappens)
    {
        break;
    }
}
```

Structured Data and Object Oriented Programming
Up until this point, all the variables we have used are single, the do not represent much of stuff, just a value. But in programming, there are certain programs that need to be solved in a different way of thinking. This is where object oriented programming begins its existence. You may have already heard of this.
In the programming we have seen, do you imagen how can you represent a book? A pen? A Car? How can they interact between themselves? It’s difficult. So, I will introduce you to the **class**.
A class is a design (or template) of something that exists in real life. Don’t worry, it’s build with the same datatypes we have seen. First, we will represent a class from a book:

```
class Book
{
public:
        string  title;
        string  author;
        int         year;
        int          pages;
};
```

The first thing is the word class followed by the name of the thing we want to describe. Inside the braces, there will be the properties of the class. Why? Let’s put is in this way: What data can we obtain from the book? Well, its title, the author, in what year was published, how many pages does it has. Notice how each property has its own data type. String for title and author (because they are characters) and integer for year and pages. After the end of the class, DO NOT FORGET THE SEMICOLON.

Now, let’s put it together in a sample program:

```
#include <iostream>
#include <string>
using namespace std;

class Book

                                        {

public:
          string  title;
          string  author;
          int              year; int         pages;

                                        };


int main()

                                        {

     Book myFavoriteBook;
     Book myLeastFavBook;

     myFavoriteBook.title = "Beginning C++";
     myLeastFavBook.title = "I dont remember";

     cout << "My favorite book is " << myFavoriteBook.title << "\n"; cout << "I really dont like " <<
myLeastFavBook.title << "\n"; cin.get();
     return 0;

                                        }
```

Here, we declare our favorite type as a data type BOOK. After this, we can access the members of that object using the “.”operator. After the point, we have access to all the properties from the object. Although both books are from the same BOOK class, each one has its own separated set of properties.

Chapter 2: Types of things to build with C++

There are many kinds of things to build using C++. As mentioned before, C++ is well preferred for its efficiency and speed of execution.

C++ is a multipurpose programming language. That means that it can be used to program literally everything. There are other languages that are "purpose specific", for example, the R programming language is for statistics, or MATLAB, that is used for math and matrixes. The kind of things that can be built using C++ literally depends on the imagination and creativity of the programmer. Here we will list some of them.

Security

Think for a moment for the antivirus you have installed (and if you don't have any installed, anyway think of it). It must be very fast for searching and inspecting files (at binary level), have firewalls, internet connectivity and a good communication with the operating system. The only language that can meet these requirements is C/C++. The features of C/C++ enables the programmer to program at low level (binary and assembly) for the most complicated work and also work at the high level (the APIs used to communicate to Windows).

Videogames

Yes, C++ may be one of the most (or probably the most) preferred languages to write videogames. Its speed at performing mathematical operations gives it a pretty nice advantage over its competitors. The runner up is C#.

And not only videogames, also some back engine motors are programmed in C++, like Unreal Engine. It is developed almost all in C++ (the rest in assembly). And, I believe that this one is even harder because, if you have a little knowledge on videogame development, Unreal Engine is used for creating videogames. Examples of them are: Tom Clancy Ranbow Six Vegas, Gears of War 2 & 3 & 4, Dead Island 2, Mass Effect 2, BioShock, Assassin's Creed Chronicles Trilogy and I am only mentioning the famous ones. And these are made using Unreal Engine.

Operating Systems

This one is very hard to explain. Imagine a full operating system: booting, libraries, peripherals, APIs, memory management, file system, command line, and graphical user interface, everything you need. Obviously, some of the lowest parts of an operating system cannot turn the back to assembly language, but the rest can be managed using only C++.

Examples: well, Windows 95, 98, 2000 and XP. Apparently, Windows Vista had both C# and C++ and that was some of the reasons of its poor effiency, after that, Microsoft used completely C#.

Another one: Mac OS X.

Another programming language

It sound kind of strange. You might think "How can another language be made using C++?" Well yes, it can be done. It is not easy but indeed can be accomplished with some programming skills and some of Compilers Theory. Guess what language was made using C++? Java.

Being a little bit more technic, the Java Virtual Machine (the box that executes everything Java need regardless the operating system) was made using C. But, C++ inherits all of the features of C plus its own STL, memory management and many more tools.

The same story goes to python. Python is an interpreted programming language and not a compilation programming language. What this means is that the code is executed by an interpreter, rather than the operating system. In python there are no executables, only scripts. But I will let you guess in what language was written the interpreter? Yes, that's right.

In conclussion, you can develop whatever comes to mind using C++, it's just a matter of creativity.

Chapter 3: Project #1 Calculator

This project will focus on the use of the math tools that C++ has for performing math calculations.

Look at this code:

```
#include <iostream> #include <cmath>
```

```
using namespace std;
```

Here we define our headers. The <cmath> header is the one that has the functions that C++ uses for the math operations. It is an original C library that can be used along with C++. Our calculator will have four kinds of operations: arithmetical, trigonometrical, exponential and logarithmic. In our main function we’ll have something like this: *int main()*

```
{
```

```
int sel = 0;
```

```
cout << "ADVANCED CALCULATOR\n"; cout << "ENTER THE TYPE OF
OPERATION YOU WANT TO CALCULATE\n"; cout << "[1] Arithmetic\n";
cout << "[2] Trigonometric\n"; cout << "[3] Exponential\n"; cout << "[4]
Logarithmic\n"; cout << "Your choice: "; cin >> sel;
```

```
switch(sel)
```

```
{
```

```
case 1:

arithmetic();

break;

case 2:

trigonometric();

break;

case 3:

exponential();

break;

case 4:

logarithmic();

break;

default:

cout << "invalid operation"; }

return 0;

}
```

Here we define our menu for the user to select the kind of operation it wants to perform. This operation is stored in the sel variable. After this, we tell the user the different kinds of operation along with the value that must be entered.

The value for sel will be used to control the flow of the program as can be seen on the switch statement. Only applies for values between 1 and 4. With something different than this the program will end.

```
void arithmetic()

{

int op = 0;

float A = 0;

float B = 0;

cout << "Select operation\n"; cout << "[1] Addition\n"; cout << "[2]
Substraction\n"; cout << "[3] Product\n"; cout << "[4] Division\n"; cin >> op;

cout << "Enter first number: "; cin >> A;

cout << "Enter second number: "; cin >> B;

cout << "Result: "; switch(op)

{

case 1:

cout << (A+B);

break;

case 2:

cout << (A-B);

break;
case 3:

cout << (A*B);

break;

case 4:

cout << (A/B);

break;

default:

cout << "Invalid operation"; break;

}

cout << endl;

}
```

In the arithmetic function, we request another time the user what kind of operation wants to perform, This section is only for the four basic operations. Here the user enters the kind of operation and both operators. After the program knows this data, it can perform the opration and show it to the screen. At the end, the control flow return to main.

```
void trigonometric()

{
```

```
int op = 0;

float val = 0.0;

cout << "Select\n"; cout << "[1] Sine\n"; cout << "[2] Cosine\n"; cout <<
"Op: ";

cin >> op;

cout << "Enter value: "; cin >> val;

if(op==1)

                                        {

cout << sin(val);

                                        }

else if(op==2)

                                        {

cout << cos(val);

                                        }

else

                                        {
```

```
cout << "Invalid operation"; }

cout << endl;

}
```

With the trigonometrical part, is almost the same, but here the user can calculate the sine and cosine functions. The user gives the value and the program return the result.

```
void exponential()

{

float base = 0.0;

float eee = 0.0;

cout << "Enter base: "; cin >> base;

cout << "Enter expnent: "; cin >> eee;

cout << pow(base, eee) << endl; }
```

The exponential part calculates any operation from x^n. Here the user enters the base and the power and, C++, using the pow() functions, calculates the value and displays it.

```
void logarithmic()

{
```

```
float value = 0.0;

cout << "Enter value for calculate the log(e): "; cin >> value;

cout << log(value) << endl; }
```

Logarithmic section is almost the same but calculates the same but here, it calculates tle e-based logarithm of the data that the user inputs to the program.

Chapter 4: Project #2: Agenda

For this project, we will simulate a phone agenda, like the one you have in your house and use them to keep record of all the important numbers you need, but we will do it using C++. For each contact, their name and their phone will be stored (although you can add as many fields as you want). Also, we will be using a new and very powerful feature of the Standard Template Library of C++: vector.

Using vectors

Imagine this phone agenda will have more than 100 phone numbers. Are you going to declare 100 variables for the numbers? And another 100 for the names? Of course not. Here is where vector fits perfectly. With vector, you will be able to insert as many elements as you like. For using vector inside C++, we must add the library <vector> in our code. We will also need iostream and string.

So, our libraries will be:

```
#include <iostream>

#include <string>

#include <vector>

using namespace std;
```

After the namespace std, we are going to declare our vector. The will go outside the main function. By declaring any variable outside main (or any function) it means that this variable will be global. In other terms, anyone has access to them for reading and writing. Look closely at the syntax:

vector<string> Names;

vector<string> Phones;

The word vector obviously says that these variables will be vectors, but now, look inside the signs.

There is the word string. With this, we tell C++ that our vectors will contain only strings. We can only have vectors of ints, of string, or any. You cannot mix data types in a single vector. Now, we will write our main function. Due to the nature of our program, we need to be able to navigate through its sections. The program will have three sections: add a contact, search for a contact using its ID and search for a contact using their name. For accomplish this, we will use a menu that will give us these three options and also an exit from the program.

The main function would like the following code: int main()

{

int sel = 0;

while(true)

{

```
cout << "My Agenda++\n\n";

cout << "Choose a number to execute an option\n\n"; cout << "[1] New
Contact\n";

cout << "[2] Search by ID\n";

cout << "[3] Search by Name\n"; cout << "[4] Exit\n";

cout << "Your choice: ";

cin >> sel;

switch(sel)

                                {

case 1:

case 2:

NewContact();

break;
SearchByID();

break;

SearchByName();

break;

case 3:

                                  }

if(sel==4)

                                  {

break;

                                  }

                                  }
```

return 0;

}

We only declared a single variable, named sel. Sel (for selection) will be used for the choice that the user selects at the menu. After the declaration, you can see how we declared an infinite loop using while(true). Why do we need an endless loop? Because the program will continue its execution until.. well, we don't know. Inside our loop, there are the cout lines. In here, we show a little title to the user, "Agenda++" followed by two line jumps. And the four following couts are used for telling the user that there are four different choices in the program. By selecting anyone of these, the program will know where to go. Next to this, we left the program to the user, where he will introduce its selections.

That selection falls into the switch statement. If the user selects 1, the program will go into the NewContact() function.

Functions In C++ are a way of modeling the program. Not all the code is written inside the main function because it would be very difficult to follow and also to imagine how all of the piece of code connect themselves. Instead, we divide the program into small modules, where each one takes care of a single task.

Important note: In the code, the main function must be the last function. Write all of the following functions in lines before main.

In this example, the function NewContact() has the job to request the user to give a new name and a new phone, like this: void NewContact()

{

string name;

```
string phone;

cout << "\n\nEnter a name for the contact: "; cin >> name;

cout << "Enter the phone for this contact: "; cin >> phone;

cout << "The ID for this contact will be " << Names.size() << "\n\n";

Names.push_back(name);

Phones.push_back(phone);

                                        }
```

Look how the function starts, it is very similar to the main function. The word void means that this function does not return anything (right now may sound confusing, but you'll get it soon). After the brace is where the function code actually starts. The first thing we do is to declare a couple of string, one named name and the other one phone. These string will be used for the input from the user.

We request the user to enter the name and the phone for the new contact. Now that we have it, we must assing this contact its ID number. Remember classes? The vector contains a property named size(), which tells us how many elements does the vector holds. In this case, we assign the ID as the current size.

After that, we use another great tool from vector, actually the most basic one: push back. Imagine a vector like a deck of cards. The function push back will add a new element at the back of the actual deck.

For this example, we add at the end of our name vector the value that the user entered. The same thing for the the phones. And now we have both values in our vectors.

Now that we have our function for inserting values, we need to retrieve them from their storage.

That is where the functions for search will be implemented. As mentioned before, we will use two types for searching values: one using the ID of the contact and the other one using the name.

For the ID, our function will be like this: void SearchByID()

```
{

int value;

cout << "\n\nEnter the ID of the contact to search: "; cin >> value;

if(value >= Names.size())

{

cout << "This ID does not exist\n\n"; return;
```

}

cout << "Information for contact " << value << "\n"; cout << "Name: " << Names[value] << "\n"; cout << "Phone: " << Phones[value] << "\n"; }

Again, first off everything, we write the function name, in this case will be SearchByID. After this, the variable value will contain the value that the user wants to search. Before making any search, we must assure that this variable is withing range from our vector. What does this means? Well, suppose we have a vector with 5 elements and the user inputs the number 8. What should we do?

We already know that our vector is 5 elements long, so, if the number that the user enters in the search is equal or greater than the size of our vector, then it will be of range.

This is implemented using the if shown on the function. If this certain condition occurs, it will show a message on our program stating that “This ID does not exist” followed by the return keyword.

The function of return is to terminate the function. In order to NOT execute any search, we must exit the function before it reaches the next code. Because of this, the return keyword is added and after the cout inside our condition, the function immediately ends and return to the main function.

If the value entered by the user is a valid one, then, it will locate the information. Notice how in the last two couts we use the “[]” operator. This is called an index and it is used to differentiate values within the vector. It goes from 0 up to size() -1.

Let’s suppose that the user enters for name the values: John, Michael, Sean. And for the phones: 111111, 333333, 555555. Then, our vectors would be like this: Names = John, Michael, Sean

Phones: 111111,333333,555555

If we want to retrieve the information, Names[0] would be John; Names[1] would be Michael and Names[2] would be Sean. The same occurs with the phones. So here, we use the ID given to retrieve these data and the show it to the screen.

Finally, we’ll go to the search function but this time we will search by the name of the contact we want to find. The code for this function looks like this: void SearchByName()

```
{

bool found = false;

string name;

cout << "\n\nEnter the name to search: "; cin >> name;

for(int i = 0; i != Names.size(); i++) {

if(Names[i] == name)

ʃ
{

cout << "Name: " << Names[i] << "\n"; cout << "Phone: " << Phones[i] << "\n";
found = true;

}

}

if(!found)

{

}

}
```

Here we declared two variables, found and name. Name will be the string that we want to look for and found is a variable of data type BOOL (those that are only true or false) and will tell us if our search was either successful or failed.

First, we request the user to enter the name to search for and after this, we will use a for loop to cycle through all of the elements. The accumulator is started, like usual, at zero. Now, notice how our condition has changed: I != Names.size()

Thanks to this little function named size(), our for loop will be able to go through all the elements until it reaches the end of itself. With this size() function, we can be completely sure that we do not go out of bound of our vector. If you do not use size(), then you will have some kind of risk looping through the vector. If this error happens, it will be on runtime and the compiler wont warn you about this, so it’s safer to use the size function.

Each lap that our loop has will compare the string at the I position in the vector with the string that the user entered. If both strings are equal, then it will show the user the name and the phone for that contact. Also, it will change the state of found to true.

After exiting the loop, it will check if the variable found is false. The exclamation sign at the start of the variable means a negation (NOT) so, if found IS false, then the flow will fall on to the block of code of the if and show the user the message “No contact was found with this name”.

After verifying the state, the function ends and the flow of the program returns to the main function.

```
cout << "No contact was found with this name!\n\n";
```

CHAPTER 5: PROJECT #3 SCHOOL NOTES SYSTEM

This project will have control of student information along with their notes stored into a file on the hard drive. Also, the program will be able to capture the notes and to generate each student's bulletin and write it to an output file (in a plain text format). The first thing for start will be defining the headers: #include <fstream>

```
#include <iostream>

#include <string>

#include <vector>

using namespace std;
```

The new header that you can note is <fstream>. It stands for "file stream" and it contains all of the functions and objects we need to manipulate files from the operating system. The rest stays the same. Now let's define the main function: int main()

```
{

int sel = 0;
```

```
while(true)

{

cout << "School Control\n\n"; cout << "[1] Add student\n"; cout << "[2] Add
notes\n"; cout << "[3] Search student\n"; cout << "[4] Read kardex\n"; cout << "
[5] Print kardex\n"; cout << "[6] Exit\n";

cout << "Choice > ";

cin >> sel;

switch(sel)

{

case 1:

AddStudent();

break;

case 2:

AddNote();
```

```
break;

case 3:

SearchStudent();

break;

case 4:

ReadKardex();

break;

case 5:

PrintKardex();

break;

case 6:

return 0;
```

```
break;

default:

break;

}

}

return 0;

}
```

This function stays almost identical to the one in our project 1. We define the menu for the different options and let the user decide which part of the program wants to follow. We define functions for add a student, add notes, search for a student, read kardex and print kardex to a txt file. Now, the definition for the addStudent part: void AddStudent()

```
{

// Temporal variable
```

```
string temp;

// Vector for holding all variables vector<string> Data;

// [0] First Name

// [1] Last Name

// [2] Phone

// [3] ID number

cout << "Enter student first name: "; cin >> temp;

Data.push_back(temp);

cout << "Enter last name: "; cin >> temp;

Data.push_back(temp);

cout << "Enter phone number: "; cin >> temp;

Data.push_back(temp);

cout << "Enter student number: "; cin >> temp;
```

```
Data.push_back(temp);

// Open the file using the ID number string fileName;

fileName = Data[3] + ".dat";

ofstream file;

file.open(fileName.c_str());

// Write the data

file << Data[0] << endl; file << Data[1] << endl; file << Data[2] << endl; file <<
Data[3] << endl; file.close();

                                        }
```

In this function, we define temp as the variable for input and a vector named Data which will contain all of the data for the new student. In the comments, you can see the index where each data is stored. After this, we use input and output instructions to make the user enter the data from the student and save it to the vector.

When we are finished, we create a new string named makeFile. This string will have the filename for the file we want to create. In this case, it will be the student number (stored in Data[3]) plus the “.dat” extension.

The ofstream object (standing for output file stream) allows us to open a file for writing mode. We open the file using the open function and inside the curly braces the file name. The c_str() function converts our string to a C-string.

Like the cout<< object, the ofstream<< also sends output information, but in this case does not go to the display, but to our output file. Remember! << operators mean OUTPUT in any stream (iostream, fstream, sstream, estc).

And finally we close the file.

```
void AddNote()

{

vector<string> Subject;

vector<string> Note;

string temp;

string student;

int z = 0;

cout << "Enter the student ID to add notes: "; cin >> student;
```

```
while(true)

{

cout << "\nEnter subject: "; cin >> temp;

Subject.push_back(temp);

cout << "Enter note: ";

cin >> temp;

Note.push_back(temp);

cout << "Do you want to continue? [0] Yes [1] No > "; cin >> z;

if(z != 0)

{

break;

}
}

string fileName;

fileName = student + ".cal";

ofstream file;

file.open(fileName.c_str());

for(int i = 0; i != Subject.size(); i++) {

}

file.close();

}
```

The AddNote function will capture the subject names and its notes for any student. We define two vectors, one for the names and the other one for the notes and a Z variable which will allow the user to enter as many subjects as he wants. First, the function asks for the student ID and after that, inside a endless loop, the user is requested to enter the subject and the note for that subject.

After this, the program asks the user for continue entering notes or exit.

When all the subjects are entered, we create a new file with the student ID plus the “.cal” extension. We open the file and write inside of it each subject and its note.

Notice how we loop through the vector and send to the file the subject and note in each position and jump a line at the end. When we finish, the file must be closed.

```
void SearchStudent()

                                        {

string number;

cout << "Insert student ID: "; cin >> number;

string fileName;

fileName = number + ".dat";

ifstream file;

file.open(fileName.c_str());
```

```
vector<string> data;

if(file.is_open())

                                        {

file << Subject[i] << " " << Note[i] << endl; string in;

while(!file.eof())

                                        {

file >> in;

data.push_back(in);

                                        }

cout << "First name: " << data[0] << endl; cout << "Last name: " << data[1] <<
endl; cout << "Phone: " << data[2] << endl; cout << "Student number: " <<
data[3] << endl; file.close();

                                        }
```

else

{

}

}

The task for the SearchStudent function is to look for a determined file with the stored data from a student. We first, request the user to enter the ID and this ID is concatenated to the ".dat” extension. Here, we use the ifstream object (input file stream). The ifstream object contains a function named is_open that indicates us if any file has been opened correctly.

If this is the case, then we must read the file. We use the string in to store the strings from the file.

The function eof() stands for “end of file” and allow us to loop through the file until it reaches the end. We store all the lines inside the data vector and after the reading is finished, we send to the display the data.

When we finish, the file must be closed. If the file could not be opened, the program will show the message “Student not found” to the display. Now let’s go to the ReadKardex() function: void ReadKardex()

{

```
{

string number;

cout << "Insert student ID: "; cin >> number;

string fileName;

fileName = number + ".cal";

ifstream file;

file.open(fileName.c_str());

vector<string> data;

if(file.is_open())

{

cout << "Student not found\n"; string in;

while(!file.eof())

{
```

```
file >> in;

data.push_back(in);

                                        }

file.close();

for(int i = 0; i != (data.size()/2); i+=2) {

cout << data[i] << "\t" << data[i+1] << endl; }

                                        }

else

                                        {

                                        }

                                        }
```

As all of this program’s functions, request for the student ID and concatenates it the “.dat” extension. Then, tries to open this file. If it is opened, then we capture word by word and add it to the data vector. When it finished, the file is closed. Now, the following for loop may look very bizarre for you. I starts at zero. The condition for this will be I different than the size of the vector dividied by two and then I will increment by two times. This loop is written this way for allowing us to read the data from the vector in pairs, as you can see on the cout line. There we read Data[i] and Data[i+1].

```
void PrintKardex()

                                        {

string number;

cout << "Insert student ID: "; cin >> number;

vector<string> data;

vector<string> notes;

string fileName1, fileName2;

fileName1 = number + ".dat";

fileName2 = number + ".cal";
```

```
ifstream dataFile, noteFile;

dataFile.open(fileName1.c_str()); noteFile.open(fileName2.c_str()); string temp;

if(dataFile.is_open())

                                        {

cout << "Student or kardex not found\n"; while(!dataFile.eof())

                                        {

dataFile >> temp;

data.push_back(temp);

                                        }

                                        }

else

                                        {
```

```
cout << "Student not found\n"; return;

                                                  }

if(noteFile.is_open())

                                                  {

while(!noteFile.eof())

                                                  {

noteFile >> temp;

notes.push_back(temp);

cout << "\n\nREAD: " << temp << endl; }

                                                  }

else
```

```
{

cout << "Kardex not found\n"; dataFile.close();

}

dataFile.close();

noteFile.close();

// Begins writing the kardex

ofstream kardex;

string fileName3;

fileName3 = "Kardex" + number + ".txt"; kardex.open(fileName3.c_str()); //
Write titles

kardex << "STUDENT KARDEX\n\n"; kardex << "NAME: " << data[1] << ", "
<< data[0] << endl; kardex << "PHONE: " << data[2] << endl; kardex << "NO
STUDENT: " << data[3] << endl << endl; kardex << "NOTES" << endl; kardex
<< "------------------------------" << endl; for(int i = 0; i <= (notes.size()/2); i+=2)
{

}
```

```
kardex.close();
```

```
cout << "Kardex generated\n\n"; }
```

Finally, the PrintKardex() function is basically a combination from the SearchStudent() and the ReadKardex(). Furthermore, it generates a file named “Kardex” + numberID + “.txt”. Inside this file, there will be a report from the student’s data and its notes.

```
kardex << notes[0] << "\t" << notes[1] << endl;
```

Chapter 6: Project #4 Sales Point

This project will have a catalog of different product that are sold in a common supermarket. All the products are stored in a file including their price. This file can be modified. The program gives the option to the user for selling products to the customers, calculating the subtotal, the tax and the final total of the whole sale. When the sale is finished, the item quantity and the total are stored on a separate file. This file stores all the sales and will be used to generate total reports from the sales. Let’s start as always, with the libraries: #include <cstdlib>

```
#include <fstream>

#include <iostream>

#include <string>

#include <vector>

using namespace std;

vector<string> products;

vector<float> prices;
```

These are all known libraries except for <cstdlib>. This is a C library and we will use the function atof, which will convert a string to a float value. We will see this a little bit later. The two global vectors will hold the name and the values (prices) for each item. Let’s go to the main function, which is very similar to the previous projects: int main()

```
{

LoadValues();

int sel = 0;

while(true)

{

cout << "SUPERMARKET SYSTEM\n\n"; cout << "[1] New client\n"; cout <<
"[2] View Catalog\n"; cout << "[3] View sales\n"; cout << "[4] Exit\n";

cout << "Select > ";

cin >> sel;

switch(sel)

{
```

```
case 1:

NewClient();

break;

case 2:

DisplayCatalog();

break;

case 3:

DisplaySales();

break;

case 4:

return 0;

break;
```

```
}

}

return 0;

}
```

Before anything, you can see that the first thing we do is call to the CallValues function. This function will load the values of the products from the file named “products.dat”. Look at the function: void LoadValues()

```
{

// The program will load a file named "products.dat"

// This information will be stored to the vectors string temp;

float tval = 0.0;

ifstream productInfo;

productInfo.open("products.dat"); if(productInfo.is_open())
```

```
{
while(productInfo >> temp) {
products.push_back(temp);
productInfo >> temp;
tval = atof(temp.c_str());
prices.push_back(tval);
}
}
else
{
loaded.\n";
}
```

}

The program opens the file “products.dat” and if it’s open, then will read all the values. The first one is inserted in products and then, reads again, this time for the price. The price is read as a cout << "Product information file not found. Data was not return; string, but we cannot make arithmetic operations with strings, so we use atof() to convert from string to float. This value is stored in tval and we insert tval in the prices vector.

Returning to main, we have our typical menu and the user is allowed to select which option wants to execute. The first option, NewClient(), will perform a new sale: void NewClient()

{

int id = 0;

int qty = 0;

int itemqty = 0;

float sum = 0.0;

cout << "NEW SALE\n";

```
cout << "INSTRUCTIONS:\n"; cout << "Enter the ID of the product. After this,
enter the quantity and press enter\n";

cout << "If you have finished, enter -1 on product and will exit\n\n";

while(true)

                                        {

cout << "Enter product ID: "; cin >> id;

if(id == -1)

                                        {

break;

                                        }

cout << "Enter quantity: "; cin >> qty;

float value = prices[id] * qty; sum = sum + value;

itemqty++;
```

```
cout << endl;

}

float tax = (sum*0.0825);

float total = (sum+(sum*0.0825)); // After the customer has entered all items cout << "\nYou have bought " << itemqty << " items\n"; cout << "Subtotal: " << sum << endl; cout << "Tax: " << tax << endl; cout << "Total: " << total << endl; float money = 0.0;

while(true)

{

menu\n\n";

cout << "Money: ";

cin >> money;

if(money >= sum)

{
```

```
cout << "Your change is " << (money-sum) << endl; cout << "Thanks for
shopping. Returning to main break;

}

cout << "Not enough money. Re-enter money\n"; }

// Write the sale to the sales file ofstream sales;

// In ofstream, ios::app is a flag that indicates that the data written

// to the output will be appended at the end of the file.

sales.open("sales.dat", ios::app); if(sales.is_open())

{

sales << itemqty << "\t" << total << endl; sales.close();

}

else
```

{

}

}

The first thing is to delare our counter variables. Id will have the id for entering items. The qty will store the number of the same items bought by the customer. Itemqty will hold the total items bought by the customer and sum will have the sum of all the prices of the products. Inside our endless loop, we request the user to enter the ID for the product. If we do not want to enter a new item and proceed to checkout, we must enter -1. Anything else will look for a item. When a item is inserted, the user must enter the quantity of the same item. We calculate the item total with the product of the value of the item and the quantity.

This loop will continue for all the items the customer wants. When he exits with the -1, then the program goes to the checkout. Calculates the tax and the total and displays these values to the screen. The program then, requests the customer to enter a quantity of money to pay. If he enters a value that is lower to the total, then it will not proceed. When the user enters a valid value, then calculates the change and opens the “sales.dat” file. Note the ios::app. This indicates that the file will not be overwritten. This value will be added at the end of the file. We will save the item quantity and the total of the sale and finally, close the file.

Now, we will look at the DisplayCatalog() function. This function will show the customer all the products with their respective price: cout << "SALES FILE NOT FOUND! NO DATA SAVED\n"; void DisplayCatalog()

{

cout << "Product catalog\n\n"; cout << "ID\tPRODUCT\tPRICE\n"; cout << "----------------------------------\n"; for(int i = 0; i != products.size(); i++) {

prices[i] << endl;

}

cout << "\n\n";

}

This one is very simple. It loops both vectors and shows on the screen the values of both vectors at the same position. The last function is DisplaySales(). This function will read our sales file and display a report from the total of sales: void DisplaySales()

{

cout << "SUPERMARTKET SALES\n\n"; cout << "Stats\n";

string temp;

int salesCount = 0;

```
int itemCount = 0;

float sumCount = 0;

float tval = 0.0;

ifstream salesFile;

salesFile.open("sales.dat");

if(salesFile.is_open())

                                        {

cout << "[" << i << "]\t" << products[i] << "\t\t" << while(salesFile >> temp)

                                        {

salesCount++;

tval = atof(temp.c_str());

itemCount += tval;
salesFile >> temp;

tval = atof(temp.c_str());

sumCount += tval;

                                        }

salesFile.close();

                                        }

else

                                        {

cout << "SALES FILE NOT FOUND! Data cannot be generated\n\n"; return;

                                        }

cout << "Total of sales: " << salesCount << endl; cout << "Total of items sold: "
<< itemCount << endl; cout << "Total earnings: " << sumCount << endl; cout
<< "\n\n";
```

```
}
```

We will initialize our counters at zero. Sales count will hold the number of sales. Item count the total of items. Sum count the total of money sales and tval will be our temporal value for reading.

We open the file “sales.dat” and we will read everything. First, read the items and convert it to float and increment sales cound by one and also itemCount by tval. After this, we read the next word, which is the money of the sale. The value is converted to float and added to sumCount.

When the loop ends, the file is closed.

Finally, we show to the output the total of this report.

Chapter 7: Project #5 Simple command line encryption system

This project is a simple encryption and decryption system. Encryption is the process of hiding information from plain text from the user. Think a little bit about the files that we have used in the other projects. All of them are plain text. You can open these files using notepad and you can see all the content without any problem.

For the studying purpose, there is no problem. However, in real systems, the information must be protected and hidden from the user. That is the objective of this project. Storing information in a different way and after this, reading and restore the real info.

Let’s look at the initial directives: *// Program that will encrypt any given file // This program uses the Map template and uses command line input #include <iostream>*

#include <fstream>

#include <cstring>

#include <string>

#include <map>

#include <vector>

using namespace std;

There are two new libraries: <cstring> and <map>. The cstring library is an original C library that we will use for string comparison. It has many more functionalities for strings, but for this case, we will be limited to only that function.

The next one, map, is a STL container. For explaining this, look at the vector. It holds multiple values using a single variable name. All of these values are different from them by the index, that number that we put inside brackets. So, what’s up with map?

Well, map is a container where the index is not a number. In map, what we store is something named pair. It consist of two internal values: key and value (in code: first and second).

Think about this like a dictionary. In the dictionary you do not look for a word using a number. You look for a definition by a word, and that is what we do with map.

map<string, string> theKey;

map<string, string> revKey;

vector<string> keys;

vector<string> values;

The first thing we declare is our map. Now inside the symbols is string,string. This means that both key and value will be strings. Our other two vectors will hold independently the keys and values.

Revkey will be the key for the decryption process.

Now, here comes the most interesting part: loading our values and populating the map!

void LoadMap()

{

// Mapping the real value vs the encrypted value keys.push_back("a"); values.push_back("0"); keys.push_back("b"); values.push_back("d"); keys.push_back("c"); values.push_back("b"); keys.push_back("d");

```
values.push_back("x"); keys.push_back("e"); values.push_back("g");
keys.push_back("f"); values.push_back("i"); keys.push_back("g");
values.push_back("z"); keys.push_back("h"); values.push_back("u");
keys.push_back("i"); values.push_back("a"); keys.push_back("j");
values.push_back("c"); keys.push_back("k"); values.push_back("y");
keys.push_back("l"); values.push_back("3"); keys.push_back("m");
values.push_back("q"); keys.push_back("n"); values.push_back("7");
keys.push_back("o"); values.push_back("f"); keys.push_back("p");
values.push_back("p"); keys.push_back("q"); values.push_back("v");
keys.push_back("r"); values.push_back("8"); keys.push_back("s");
values.push_back("e"); keys.push_back("t"); values.push_back("j");
keys.push_back("u"); values.push_back("5"); keys.push_back("v");
values.push_back("h"); keys.push_back("w"); values.push_back("r");
keys.push_back("x"); values.push_back("2"); keys.push_back("y");
values.push_back("m"); keys.push_back("z"); values.push_back("n");
keys.push_back(" "); values.push_back(" "); for (int i = 0; i != keys.size(); i++)
{

pair<string, string> a;

a.first = keys[i];

a.second = values[i];

theKey.insert(a);

a.first = values[i];

a.second = keys[i];

revKey.insert(a);


                                   }



                                   }
```

This code is a little bit tedious. Here it is the key of our encryption. The first row, used for keys, will be our original values. The second row will be our encrypted values. That means, every letter ‘a’ will be changed for a ‘0’. Every letter ‘t’ will be changed for a ‘j’ and so on with the rest of the letters.

In this code I did not inserted capital letters. So, our message can only contain small letters and spaces. Everything else will not be encrypted. But as you can see, it is very easy to enter more values inside the vector.

Then, we have our loop. It will go from zero to the size of our keys. Inside, we declare a new PAIR of values, both of type string. The pair name is ‘a’.

The value for first (the key) will be our key string at position I and the value for our second (the actual value) will be inside the I position of values. After we create our pair, it will be inserted into the map using the insert() function. The same goes for the decryption key in revkey, but here the values are inverse.

Now, the main function will be different than before. Take a very close look to the following piece of code: *int main(int argc, char* argv[])*

{

// argc - number of parameters entered through command line // argv - parameter entered from command line // argv[1] - file to process

// argv[2] -

// -e Encryption

// -d Decryption

cout << "Loading data.\n";

LoadMap();

vector<string> inputData;

```
string message;

string messageEnc;

string outputPath;

if (!strcmp(argv[2], "-e"))

                                        {


                                        }

else if (!strcmp(argv[2], "-d"))

                                        {


                                        }

outputPath += argv[1];

ifstream inputFile;

ofstream outputFile;

cout << "Opening files.\n";

inputFile.open(argv[1]);

outputFile.open(outputPath.c_str()); if (inputFile.is_open())
```

```
                                   {

outputPath = "ENCRYPTED ";

outputPath = "DECRYPTED ";

// Obtain the input

cout << "Reading files.\n";

inputFile >> message;

// For each letter in the input, generate its //  corresponding encrypted value

for (int i = 0; i != message.size(); i++) {

cout << "Encrypting.\n";

string tempChar;

tempChar = message.at(i);

// Differentiate between encrypting and decrypting if (!strcmp(argv[2], "-e"))

                                   {


                                   }

else if (!strcmp(argv[2], "-d"))

                                   {


                                   }
}

else

{

}

}

// Generate the output

cout << "Writing encrypted file.\n"; outputFile << messageEnc;

inputFile.close();

outputFile.close();

messageEnc += theKey[tempChar];

messageEnc += revKey[tempChar];

}

else

{

}
```

cout << "Process finished. Output file is " << outputPath << ".\n";

cin.get();

return 0;

}

What in the world is argc and argv? Well, you may have heard about command line. In the command line you can execute programs in a ver fast interface and have the option to configure these programs. This particular program is designed only to work in command line. An argument is a value that is passes through the command line and it goes inside our C++ program. The variable argc is the count of arguments that our program has received. By default, is always one: the program name. The second parameter, argv, has the actual values that the user enters to the program.

This program will receive two arguments:

- The file to encrypt/decrypt

- The mode for processing

cout << "File " << argv[1] << " not found.\n"; In the folder where you have your exe, create a new file named “data.txt” and write something, for example “hello” and save it. After this, go to the settings of your project and look for something called “Executable to Run/Debug”. After “./$(ProjectName)” enter: data.txt –e These represent the file you want to encrypt and –e means that it will perform an encryption. This option allows the compiler to automatically insert parameter to your program without having to go all through command line. But now let’s return to the code.

We load our maps using LoadMap() function and a vector named input data. This one will have all the stuff that is inside the file. Message will have the total message. MessageEnc will have the transformed message. OutputPath will have the name of the file where the encrypted/decrypted message will be written.

After this, we will si an if structure. Strcmp is a string comparison. If the second argument (argv[2]) is “-e” then it means that is is an encryption. So, the filename will have the “ENCRYPTED” label. If it has the “-d” string, then it will be a decryption.

Then, we proceed to the file processing. Here we will be using both input and output file streams.

The input has our original message and the output the transformed message. We open the input and read all the message.

After this, we will treat our string like if it was a vector. For each character inside the string, will find its counterpart on the maps for its translation. However, the [ ] operator are not allowed in a string. In order to perform an index lecture, we must use the at() function. The actual character will be stored into the tempChar variable.

When we have this character, we run again the comparison. If the second argument is “-e” it means that it is an encryption. So, we encrypt.

We will append at the end of the messageEnc the VALUE corresponding to its KEY. So, the KEY will be the letter and the value will be the value associated with that key. All from the theKey map.

If it is a decryption process, then, instead of comparing with theKey, we will compare it to the revKey map.

When it finishes with all the characters, a message appears in out console and writes to our output file the processed message. Finally, both files are closed. If the input file is not found, then it will show an error message.

The program does not show any message. Everything goes to the files. Finally we finish the program and return the main function.

CHAPTER 8: PROJECT #6 DUEL GAME

This project is a simple demonstration of an automated videogame, everything running through console. The main idea is a duel. A confrontation from cowboy vs cowboy. There are three different kinds of cowboys:

- Basic
- Resistence
- Fast Attack

Where the Basic has normal values, the resistance has lower attack point but more life points and the fast attack has more attack points but less life points. The user will enter the kind of cowboy that he wants. After this, the machine will randomly select any of the three. In this program, we will use classes for defining our types of cowboys. Let's look at the headers: #include <iostream>

```
#include <cstdlib>

#include <ctime>

using namespace std;
```

The new header in this project is <ctime>. Ctime is an original C header and will be used to generate random numbers. Now, let's see our classes: c lass cowboy {

```
public:
```

```
int life;

int player;

int maxatk;

int minatk;

int type;

cowboy()

                                        {

                                        }

cowboy(int v, int am, int an, int t) {

life = v;

maxatk = am;

minatk = an;
```

type = t;

}

};

The cowboy class is our definition for Cowboy. As you can notice, it contains its fundamental values: life points, the player, the maximum attack points, the minimum attack points and the type of cowboy.

Its constructor, cowboy(int, int, int, int) is a function that will let us assign the four values in a single instruction when the cowboy is created. Before going into our Duel class, let’s look the main function: int main()

{

duel theDuel;

theDuel.mainmenu();

system("pause");

return 0;

```
                                   }
```

In our main function, we only define an instance of our Duel class and we start the duel with the main menu function from Duel. Here, the flow control is inside the duel, and not main.

```
class duel

                                   {

public:

duel()

                                   {

                                   }

void mainmenu()

                                   {

int choice;
```

```
do

                                    {

cout << "Main Menu\n\n"; cout << "1-Play\n";

cout << "2-History\n"; cout << "3-Exit\n\n"; cout << "Select: ";

cin >> choice;

if(choice==1)

                                    {

                                    }

else if(choice==2)

                                    {

                                    }

                                    }
```

```
while(choice!=3);

}
```

Up until here we define the Main Menu of the duel as if we were on the main menu on other projects. This is another way of working around with menus.

```
void selectcowboy()

{

cout << "\n\nSelect your cowboy:\n\n"; cout << "1-Basic\n";

cout << "2-Resistence\n"; cout << "3-Fast Cowboy\n\n"; cout << "Choose: ";

int choice;

cin >> choice;

cowboy player;

cowboy machine;
```

```
if(choice==1)

                              {

player.life = 30;

player.maxatk = 15;

player.minatk = 8;

player.player = 1;

player.type = 1;

                              }

else if(choice==2)

                              {

player.life = 50;

player.maxatk = 15;
player.minatk = 8;

player.player = 1;

player.type = 2;

                                        }

else if(choice==3)

                                        {

player.life = 28;

player.maxatk=9;

player.minatk=5;

player.player = 1;

player.type = 3;

                                        }
```

```
int rival = rand()%3 + 1;

if(rival==1)

                                        {

machine.life = 30;

machine.maxatk = 15;

machine.minatk = 8;

machine.player = 1;

machine.type = 1;

                                        }

else if(rival==2)

<< machine.life << "\n\n"; {

machine.life = 50;
machine.maxatk = 15;

machine.minatk = 8;

machine.player = 1;

machine.type = 2;

                                        }

else if(rival==3)

                                        {

machine.life = 28;

machine.maxatk=9;

machine.minatk=5;

machine.player = 1;

machine.type = 3;
```

```
}

else

{

cout << "Error. Choice = " << choice << "\n"; }

// Inicia el duelo

cout << "\n\nPreparing duel\n\n"; cout << "Player life: " << player.life <<
"\nMachine life: "

cout << "Duel starts now!\n\n"; bool s = true;

while(true)

{

if(s==true)

{

player.minatk)+player.minatk; points\n";
```

```
}

else

{

machine.minatk)+machine.minatk; points\n";

"\n";

"\n";

}

cout << "Player's remaining life: " << player.life << cout << "Machine's
remaining life: " << machine.life << if(player.life <= 0)

{

}

else if(machine.life <= 0) {
                                   }
if(s==true)
                                   {
                                   }
else
                                   {
                                   }
                                   }
                                   }
void story()
                                   {
}

};
```

Let’s explain the code inside SelectCowboy(). Yes, it is very long, but let’s look at it very closely.

The first thing is that we request to the player to select what kind of cowboy does he want. Basic, resistance or Fast.

After this, we create two cowboys, one for the player and another one for the machine. Then comes an if instruction. If the user select the basic cowbow, you can see the parameter that will have the cowbow. These values are the ones that change in each type of cowboy.

After this, we define the type of cowboy that our rival will have using the rand() function. This function is included in the ctime header. The same configuration parameters are applied for the cowboy that the machine will use in the duel.

When we finish configuring this parameters, finally, the duel starts. The s variable, which is a bool variable, will indicate us the turn. If s is true, then the turn is for the player. If the s is false, then it’s the machine’s turn.

Entering the player’s turn code, the attack that the player will have will be a random number between its max attack and its min attack. Due to the configuration options, if the player has the fast attack option, then the attack will be doubled. However, if the machine has the resistence type, the attack will be divided by three.

After this, the prompt will show with how many points the attack was made and these attack points will be substracted from the machine’s life point. This same process happens when it’s turn for the machine to attack. In this case, the player’s life points are decreased in relationship with the machine’s attack points.

After both players attack, we show on the screen the remaining life points for each player and then evalueate wether both players are still alive. If any of the two has zero or negative life points, the game ends and shows a message where the player lost or won. If none of this happens, then the value of ‘s’ will be inverted and the loop will star all over until a player loses all of their life points.

Finally, the story function is only an add-on for the game. You can replace these string with anything that you wish.

```
cout << "Here you can insert any history\n"; cout << "Fill this part\n";
```

CHAPTER 9: PROJECT #7 POKER GAME

In this final project, we will simulate a full OOP Poker Game. Here, we will use multiple classes. So,let’s get started.

First of everything, we must define our class for any card. Here we will work with multiple files, so, create a new file named card.h and add this code: *#ifndef CARDH_*

*#define CARDH_*

class Card

{

public:

int number;

int sign;

void Assign(int, int);

};

#endif

When we use multiple files, the #ifndef is a pre-processor definition. With this, we assure that this file will only be included once. If it is included multiple times in the same program, will cause trouble for duplicated symbols.

So, our card will have its number, its sign and a function named Assign, and with this function, we will assign each card its values. Now, create a new file named “card.cpp”:

```
#include "card.h"

void Card::Assign(int num, int si) {

number = num;

sign = si;


}
```

In this file, there will be the implementations of our functions. Here, we assign the parameters from the function to the values that the object will have. Notice that we must include our “card.h” file. That is the one that we created before. It must be included so it can implement the functions.

Now that we have the card classs, we must create the player class. Player.h defines like:

```
#ifndef PLAYERH_

#define PLAYERH_

#include <iostream>

#include <string>

#include <vector>

#include "card.h"

using namespace std;

class Player

{
```

```
public:

string name;

vector<Card> hand;

void GetCard(Card);

void ShowHand();

                          };

#endif
```

The Player class will have two properties: the name of the player and a vector. Look: here we do not use any string. It is a CARD! The class we defined previously. This vector of cards can be viewed like the hand that the player owns. Also, we have a function named GetCard(), this function will allow our player to receive cards and add them to its hand. And finally, the function ShowHand().

This will only show the values inside the hand. Let's see the implementation that will be inside *"Player.cpp":*

```
#include <vector>

#include "card.h"

#include "player.h"

using namespace std;

void Player::GetCard(Card c)

                          {
```

```
hand.push_back(c);

}

void Player::ShowHand()

{

cout << name << "'s hand: "; for (int i = 0; i != hand.size(); i++) {

cout << hand[i].sign << "-" << hand[i].number << " "; }

cout << endl;

}
```

In this implementation, we must include Card and Player because we will be using both. The function Card receives a parameter of type Card and is inserted in the hand. The show hand loops through the hand of the player, obtains the values for each card and displays them to the screen.

But… who will give the cards? The dealer obviously!

Let’s define a class for our dealer in “dealer.h”: *#ifndef DEALERH_*

```
#define DEALERH_

#include <string>

#include <vector>

#include "card.h"
```

```
using namespace std;

class Dealer

                                   {

public:

vector<Card> Deck;

Dealer();

Card GiveCard();

void ShuffleCards();

                                   };

#endif
```

The dealer only has one property but it is the most important property from the game: The deck!

And how will we represent de deck? Also as an vector of cards. The dealer will also have the hability to give cards and to shuffle the cards. Now, lets look the implementation: *#include <algorithm>*

```
#include <vector>

#include <string>

#include <random>

#include <chrono>
```

```
#include "dealer.h"

#include "card.h"

using namespace std;

Card Dealer::GiveCard()

                                                    {

Card c = Deck[0];

Deck.erase(Deck.begin());

return c;

                                                    }

void Dealer::ShuffleCards()

                                                    {

unsigned seed =

std::chrono::system_clock::now().time_since_epoch().count();
shuffle(Deck.begin(), Deck.end(), std::default_random_engine(seed)); }

Dealer::Dealer()

                                                    {
```

```
for (int i = 0; i != 12; i++) {

for (int j = 0; j != 4; j++)

                                        {

Card newCard;

newCard.number = (i + 1);

newCard.sign = (j + 1);

Deck.push_back(newCard);

                                        }

                                        }

                                        }
```

Here we inserted new libraries:

- Algorithm: will be used to shuffle our deck
- Random: generate a random number
- Chrono: access to timing libraries

First, look at the GiveCard(). It will select the card that is on the top of the deck. Will remove it form them and then extract it to another part, that would be the hand of any player. On the ShuffleCards(). We will generate a time-based seed which we will use for shuffling the deck in a random order. Also, look at the

Dealer() function. It has a nested loop (a loop inside another loop) and, with this, we generate the cards. From 1 to 12 and each from 1 to 4. After this, each card is inserted to the deck and now the deck is created!

Now, the most important part, the main function: *#include <iostream>*

```
#include <string>

#include <vector>

#include "player.h"

#include "card.h"

#include "dealer.h"

using namespace std;

// Number of cards allowed in hand int CardNumber = 5;

int main()

{

// Create players

Player player1;

Player player2;

player1.name = "CPU";

string name;

cout << "Enter your name: "; cin >> name;

player2.name = name;
Dealer theDealer;

theDealer.ShuffleCards();

// Give cards to the players

for (int i = 0; i != CardNumber; i++) {

player1.GetCard(theDealer.GiveCard());
player2.GetCard(theDealer.GiveCard()); }

player2.ShowHand();

cout << "\n";

player1.ShowHand();

return 0;


                                        }
```

There are no unknown libraries. Remember that we must add all of our class headers. There is the constant CardNumber. It will be used by the dealer for giving cards. When the main function starts, two players are created. We assign player1 the name “CPU” and for player 2, the name will be given by the user.

After this we create the dealer (and by creating the dealer, also the deck is created) and tell the dealer to shuffle cards. Now, from 0 to CardNumber, the dealer will obtain a card for each player, remove it from the deck and give it to the hand of the player. After this, both hands are shown and the outcome of the game can be determined.

In this example, it can be very easy to understand how all of these real life objects are abstracted and taken to a programming paradigm like C++. Here there is no functional logic, only OOP logic.

Chapter 10: What to do next

Up until now, we covered some basic and intermediate stuff concerning C++. However, in order to master C++, there are more specialized topics that should be learned.

Memory management: The use of the memory available from the operating system into our objects. The keywords new and delete. With these two keywords, you can manage how many memory is being used, assign when and to whom assign memory and even optimize your programs.

Standard Template Library: In this course we covered only two templates: map and vector. However, the C++'s STL is way beyond this these two objects. Structures like the stack, queue, list, deque and more are built-in within C++.

Iterators: The iterators is the way that C++ loops through the templates but do not use indexes. Instead, the iterators adapt to the needed container.

Pointers: Pointers are a very important part of C++ and will allow you to have a better memory management and create multiple objects saving huge amounts of memory.

String management: Multiple operations can be realized using strings. Search for patterns, complex concatenation, tokenize tokens, processing strings, replacing totally or partially strings, *etc*.

Fundaments of OOP: Although we covered some points from OOP in the sample projects, this course was not about OOP. So it would be a good idea to cover full

OOP theory in order to generate better design of programming. Some concepts would be class, abstraction, member, member functions, simple inherence, multiple inherence and even polymorphism.

Create your own libraries: As we saw on the course, you can generate your own .h and .cpp files. Here we used them as external linking for classes. But it can be also implemented with your own home-made functions. You might have functions that you need to use multiple times and do not want to have to rewrite them several times. So, a good idea would be to implement all of these functions in a single .cpp file and add it to a future project.

If you cover all of points mentioned here, you can move on to more advanced features like:

- Templates
- Operator overloading
- Complex file processing
- Graphical programming
- On windows: Microsoft Foundation Classes
- On Linux: GTK+ or QT

Even, with graphical programming, there is the OpenGL open source programming. With OpenGL you can generate 2D and 3D graphics using C++. In the 2D mode, you can draw multiple figures using vertex and matrix. You can add textures, colors and refresh screen whenever you want. In the 3D mode, you can design your own 3D environments, define textures and 3D models. At this point, animations become accessible. Also, in both modes, you can enable the use of the keyboard and assign actions with each key. With OpenGL you can also design videogames, from the easiest to the hardest. It depends on your programming skills. Moreover, OpenGL also enables you to add background audios, and all of this using C++. However, you should take note that OpenGL runs well on the Microsoft Visual Studio compiler. Extra configurations might be needed in the GNU G++ compiler. As you can see, the limit in C++ programming is up to your imagination.

CONCLUSION:

After the topics we have covered, you can have a clearer scene of how C++ programming is developed. We have tried to have an applied aspect of the applications of C++. The reason is that many times, the user does not understand how all the pieces can work together as a complete system. So we have been careful in the applied focus for this course.

As you might have noticed, the complex programs are, at the end, are made using the same basic instructions explained in chapter one. One example of this is the creation of a menu: here we have variable declaration, an infinite loop, a switch statement and some input. Using these basic structures we can build the menu and many more stuff.

One thing that we have to remember (and that many people don't see) is that the console is not the only output. You might think that the console is completely useless and is not attractive. And, in a certain way, this is right. The console is not the appropriate input and output system for most programming in these days. However, remember: the power of C++ is its speed, its complexity and its STL. The input and output system can be always changed: sometimes it can be from console, other times can be a file, and another a graphical user interface (GUI) or whatever you want. But don't be mistaken by the false fact that the programs that are ugly are completely useless.

Having this course, you can advance to a more complex graphical system, where you can make more comfortable software while still having the power of C++. Examples of this is GTK+ or QT. With these C++ expansions, you can design graphical interfaces for complementing your projects.

This course also covered some features that C++ inherits from its predecessor, C, like the math functions. These are not the only ones. You can dive up into the complete C library and find a bunch more of tools to use in your C++ programming, just remember to make the proper type conversions.

After this course, you are now able to develop your own solutions. And you are not limited by the solutions. Now, you can do file processing, data storage, report creation, data type conversion and more features that are widely used for calculations. Something has to be said: plain text is not the best way to save information. The optimal way would be a database. But this is a more complex to cover in a basic course. However, with this basic knowledge of data storage, you can move on to a better storing technique.

After this point, it is in the interest of this course that you are now able to interpret C++ code and to provide solutions using the language. Many resources are available on the internet, like reference pages, that can complement all of your C++ knowledge and that you might now, with the concepts that we have covered, understand an implement new functionalities. Now the question would be, what are you going to do with your newly acquired knowledge?

Did you enjoy this book?

I want to thank you for purchasing and reading this book. I really hope you got a lot out of it.

Can I ask a quick favor though?

If you enjoyed this book I would really appreciate it if you could leave me a positive review on Amazon.

I love getting feedback from my customers and reviews on Amazon really do make a difference. I read all my reviews and would really appreciate your thoughts.

Thanks so much.

W.B Ean

p.s. You can click here to go directly to the book on Amazon and leave your review.